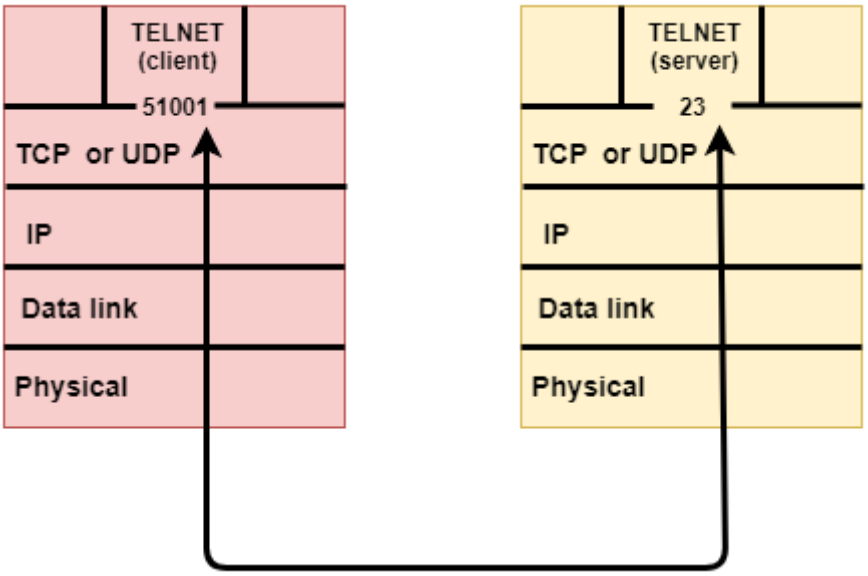
Transport Layer

The task of the **transport layer** is to provide services to the application layer above it. To achieve this, it uses the services provided by the layer below it, Layer 3, or the network layer.

The purpose of the transport layer is to provide a logical **end-to-end communication channel**. In particular, it can compensate for limitations in data transfer by implementing functions that improve reliability and provide guarantees for the communication.

The transport layer is found only in **end systems**, or hosts, and provides logical communication between application processes.



The image shows the transport layers of two hosts communicating with each other. To communicate, however, their data must pass through the lower layers

Transport Protocols

The transport layer provides the application layer with communication services offering specific characteristics, mainly through the **TCP**, **UDP** and **QUIC** protocols.

TCP

TCP, or **Transmission Control Protocol**, is one of the main communication protocols used at the transport layer.

TCP provides a **connection-oriented and reliable service**.

Through TCP, a reliable, bidirectional logical connection is established between the sender and the recipient.

The main characteristics of TCP include:

- **Reliability:** TCP attempts to guarantee reliable data delivery. It uses acknowledgements, abbreviated as **ACKs**, error checking and retransmission mechanisms to ensure that data is delivered correctly and in the correct order. If data is lost or corrupted during transmission, the sender retransmits it until the recipient confirms that it has been received.
- **Flow control:** TCP prevents the sender from transmitting data faster than the recipient can receive and process it.
- **Congestion control:** TCP adjusts the transmission rate according to network conditions to avoid overloading connections and intermediate routers.
- **Connection-oriented communication:** TCP establishes a connection between the sender and recipient before data transmission begins.

To transmit data, TCP divides the message received from the application layer into smaller units called **segments**.

These segments contain:

- The original data, or **payload**.
- A **TCP header** containing information such as the sequence number, acknowledgement number, control information and other parameters required to ensure correct delivery and manage data flow.

TCP segments are placed inside IP packets, which are created at the network layer, and then transmitted across the network. The IP packets may follow different routes before reaching their destination.

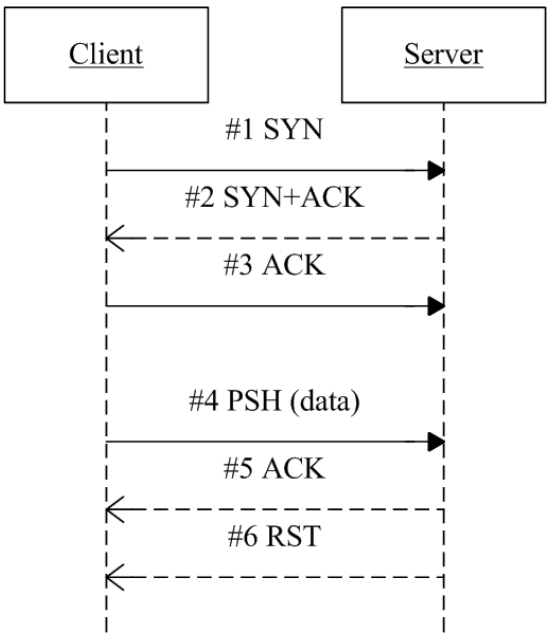
Once they arrive, the receiver reassembles the segments according to their sequence numbers and uses the control information to acknowledge their receipt.

If data is not received correctly, TCP requests that it be retransmitted by the sender to preserve data integrity and reliability.

TCP has traditionally been, and continues to be, widely used on the Internet, especially for services that require reliable communication, such as file transfers, many web communications, email and remote access.

A TCP connection goes through at least the following stages:

- **SYN:** The client requests that a connection be opened by sending a SYN message: “Start connection”.
- **SYN-ACK / ACK:** The server responds with a SYN-ACK, and the client confirms with an ACK: “Connection established”.
- **Data transfer / PSH:** The two devices exchange data through segments and acknowledgements: “Connection active”.
- **FIN / RST:** One of the devices closes or terminates the connection using FIN or RST: “Connection terminated”.



UDP

Like TCP, **UDP** is a transport-layer protocol, but it provides a very different service.

Unlike TCP, which provides reliable, connection-oriented communication, UDP is **connectionless** and does not guarantee reliable data delivery.

This means that when an application uses UDP to send data to another application, the data is transmitted without any guarantee that it will be received or delivered correctly.

UDP therefore provides a **connectionless and unreliable service**.

The main characteristics of UDP are:

- **Connectionless communication:** No connection is established between the sender and recipient before data is sent. Data is transmitted without any preliminary negotiation.
- **Fast and lightweight communication:** Since UDP does not include the connection-management and reliability mechanisms used by TCP, it is faster and introduces less overhead.
- **No flow or congestion control:** UDP does not directly provide complex reliability mechanisms such as acknowledgements, retransmissions or data reordering.
- **Commonly used for low-latency applications:** UDP is often used when speed and low latency are more important than reliability, such as multimedia streaming, online gaming and real-time communication.

Since UDP does not guarantee reliable data delivery, applications using it must manage any data loss or errors at the application layer when necessary.

UDP is used in situations where speed and simplicity are more important than guaranteed delivery. Common uses include:

- **Audio and video streams:** UDP is often used for real-time audio and video, such as videoconferencing, Voice over IP and low-latency live broadcasts.
- **Online games:** UDP is often preferred for transmitting real-time information about players' actions. Low latency is essential for providing a responsive and fluid gaming experience.
- **Real-time broadcasts:** In applications such as live sporting events, UDP may be used to provide real-time playback with minimal delay.
- **Local device-discovery services:** Some protocols use UDP to locate devices quickly within a local network.
- **Internet of Things applications:** UDP may be used in certain IoT scenarios, such as temperature sensors, where devices must transmit small quantities of data quickly and with limited complexity.

The unit of data sent by UDP is called a **datagram**.

A UDP datagram contains a header with essential control information and a payload containing the data received from the application.

QUIC

Having examined TCP and UDP, we can introduce a more recent protocol: **QUIC**.

QUIC is a modern transport protocol designed to make Internet communication faster, more secure and more efficient.

It is particularly important because it is used by **HTTP/3**, one of the most recent versions of the HTTP protocol used on the Web.

An interesting characteristic of QUIC is that it operates on top of UDP.

This may appear unusual because UDP is a connectionless and unreliable protocol.

However, QUIC uses UDP as its foundation and adds many of the functions normally associated with TCP, including reliability, congestion control and connection management.

One reason QUIC was developed was to reduce the time required to begin communication.

With TCP, a connection must first be established. When HTTPS is used, an additional cryptographic negotiation is also required.

This may involve several exchanges before the actual application data can begin to be transmitted.

QUIC integrates security directly into the protocol and uses mechanisms based on **TLS**. This reduces the number of steps required to establish a secure connection.

As a result, communication can begin more quickly in many situations.

Another important feature of QUIC is its improved management of interruptions and network changes.

For example, a smartphone may begin browsing over Wi-Fi and then switch to a mobile network.

With a traditional TCP-based connection, this change may interrupt the communication because the device's network address, or IP address, changes.

QUIC is designed to handle these changes more effectively and maintain active communication whenever possible.

QUIC also improves the management of multiple data streams within the same connection.

A modern web page does not consist of a single file: it may contain text, images, style sheets, scripts, videos and many other resources.

QUIC can manage multiple streams in parallel, reducing the risk that the loss or delay of one part of the data will unnecessarily block the others.

This is important because some packets may arrive late or be lost during network communication.

With some traditional systems, the loss of one packet may delay data that is already available. QUIC attempts to reduce this problem by separating the different communication streams more effectively.

QUIC can therefore be viewed as a protocol that combines some positive characteristics of TCP and UDP:

- Like UDP, it is built on datagrams and does not require the same rigid structure as TCP.
- Like TCP, it can provide reliability, congestion control and ordered data management. Unlike traditional TCP, however, it integrates security and can establish protected communication more quickly.

QUIC is well suited to the modern Web, where speed, security and the ability to manage many data streams simultaneously are required.

QUIC is particularly important because it forms the basis of **HTTP/3**.

Ports

In a networked system, a **port** is a logical number used to identify a specific communication channel within a host.

Ports are fundamental because they allow several applications on the same host to use the network simultaneously without interfering with one another.

Imagine the telephone switchboard of a large company.

All departments can be reached through a single telephone number—the computer's network address—but the caller can then be connected to different extensions, such as sales, customer support or administration.

The main telephone number is therefore comparable to the computer's address, while the extensions are comparable to its ports.

To summarise, ports allow a host to maintain multiple, distinct communications with other hosts at the same time. They ensure that data belonging to a specific service is delivered to the process waiting for it.

In TCP- or UDP-based communication, ports are used to associate incoming data with the correct application process.

A network communication normally involves two ports:

- The **destination port** identifies the service to which the device wants to connect. For example, an HTTPS web server normally listens on port 443.
- The **source port** is selected by the device that initiates the communication. It is often a temporary port assigned automatically by the operating system.

This allows a computer to distinguish between several simultaneous communications.

For example, when a browser opens multiple connections to websites, the operating system can use different source ports to identify each communication.

Ports can be classified into three groups:

Well-Known Ports

Well-known ports have numbers below 1024 and are generally used by operating-system services or system processes.

Server applications operating at the application layer commonly listen on these ports.

Examples include:

- **FTP**, the File Transfer Protocol, used by applications such as FileZilla: port **21**.
- **SSH**, a protocol used to establish an encrypted remote session: port **22**.
- **SMTP**, a standard protocol for sending email: port **25**.
- **HTTP**, the main application-layer protocol used to transmit information on the Web: port **80**.
- **HTTPS**, HTTP with additional security: port **443**.

Registered Ports

Registered ports are assigned to particular protocols, applications or services, but they are not reserved as strictly as well-known ports.

They are often used by less common applications and services or by newer protocols that require a standardised port number without needing the widespread recognition associated with well-known ports.

For example, custom communication protocols or specialised applications may use registered ports.

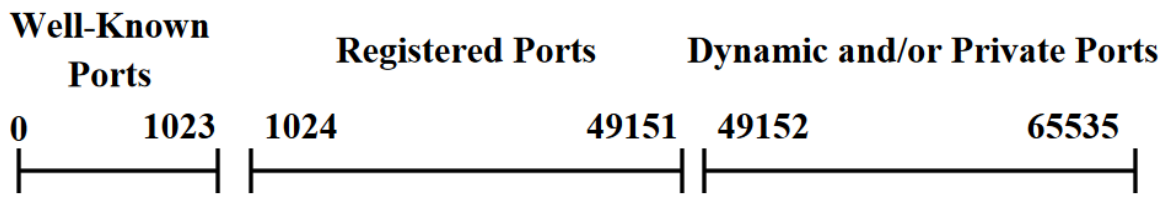
Dynamic or Private Ports

Dynamic or private ports can generally be used freely by user applications, provided that they are not already being used by another process.

They are normally assigned dynamically by the operating system and are not permanently reserved for specific protocols.

This provides greater flexibility in allocating communication resources and prevents conflicts when several applications communicate simultaneously.

-



Example

Example of a Well-Known Port

When you visit a website such as `www.example.com`, the browser may communicate with the server through the HTTP protocol using its default port, port 80.

HTTP is used to request and transfer web pages and other content over the Web.

Example of a Registered Port

Suppose a company creates a web service that adds two numbers.

The service receives two numbers as input and returns their sum as output.

The calculator program is made available on port 3000.

Clients wishing to use the service must connect to the server's address and specify port 3000. Their message must contain two numbers, and they must be able to process the returned value containing their sum. These rules form the service's protocol.

Example of a Dynamic Port

When a user opens an HTTPS website, the browser connects to port 443 on the web server.

However, the user's computer also uses a temporary source port, selected automatically by the operating system, to identify that particular communication.

For example:

Client: port 52341 → Server: port 443

If the user opens several browser tabs or services at the same time, the operating system can assign different dynamic ports to distinguish between the active communications.

Sockets

When two devices communicate through a network, knowing their addresses and ports is not enough. A mechanism is also required to manage the actual communication between the two programs.

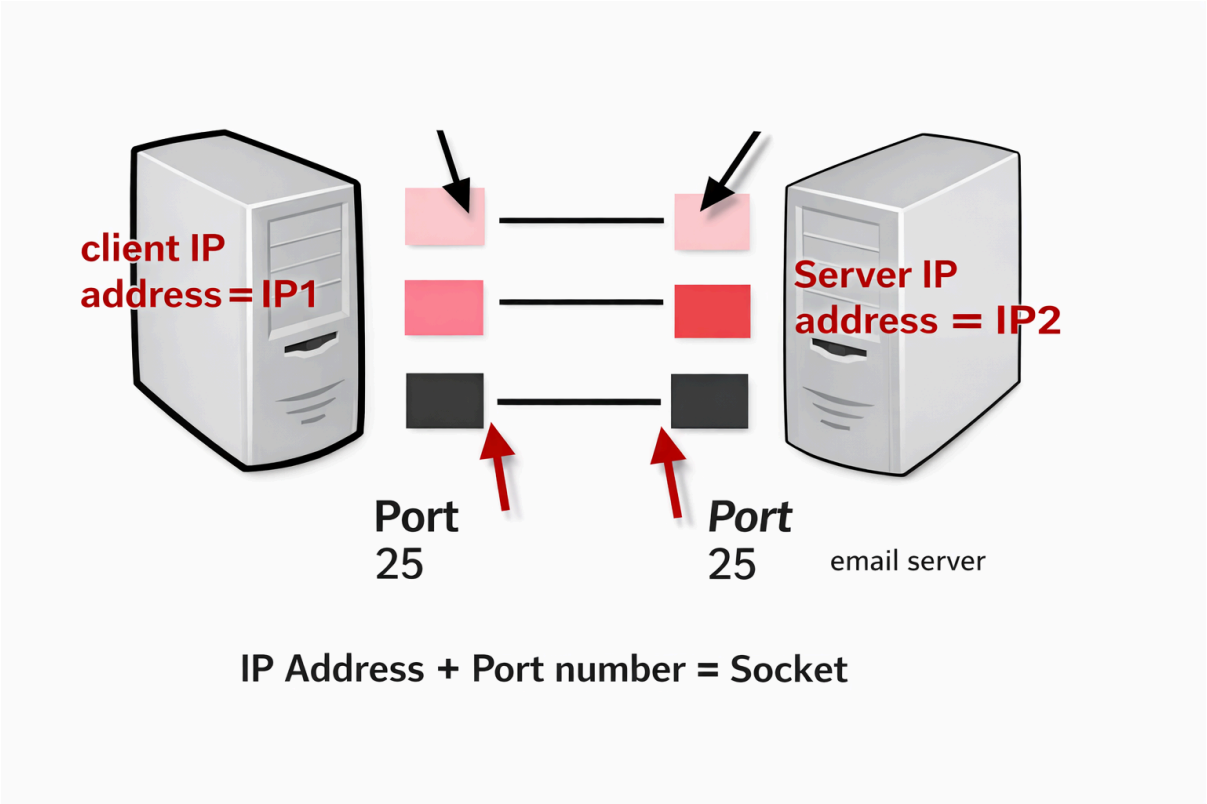
This mechanism is called a **socket**.

A socket is the software endpoint through which a program communicates over a network.

- With TCP, a socket may represent an established connection.
- With UDP, it represents an endpoint for sending and receiving datagrams.

A socket is therefore a software object that allows a program to send and receive data over a network using an address and a port:

- The address identifies the computer.
- The port identifies the program or service.
- The socket provides the software endpoint used for communication.



When an application needs to communicate with a remote application, it creates a socket:

- When using TCP, it establishes a connection with the server.
- When using UDP, it sends datagrams to a destination address and port.

Communication between the two applications takes place through their respective sockets.

A local socket is identified by at least:

- An address.
- A port.
- A transport protocol, such as TCP or UDP.

In the case of a TCP connection, communication is identified more precisely through the combination of:

- The sender's address and port.
- The recipient's address and port.
- The transport protocol being used.

Together, these elements identify a specific communication between two hosts.

The stages involved in socket communication are:

- **Creation:** The program creates a socket and associates it with an address and port.
- **Connection, for TCP:**
    - If the host is a client, it locates the server and establishes a connection.

    - If the host is a server, it listens for incoming requests. When a client connects, the server creates a new socket for that connection, containing the client's and server's addresses and ports.
- **Transmission:** Data can be sent and received through the socket.
- **Closure:** When communication has ended, the socket is closed.

Sockets can be divided into two main categories, each providing a different type of communication:

- **Datagram socket, or UDP socket:** This type of socket uses UDP. Data is transmitted through individual datagrams without guaranteeing their arrival order or successful delivery. The client and server do not establish a persistent connection; the client sends data directly to the server whenever necessary.
- **Stream socket, or TCP socket:** This type uses TCP and therefore provides connection-oriented communication, greater reliability and additional controls. However, it introduces more overhead than a UDP socket.
-

Example
The following steps take place when a user opens `www.wikipedia.org` in a browser:

- The user enters `www.wikipedia.org` into the browser.
- The system uses DNS to obtain the address of the server associated with that name.
- The browser opens a TCP socket to port 443 of the HTTPS server at the address that has been found.
- The server responds, and data is exchanged through the socket.
- The browser displays the web page.
- Once the exchange has ended, the socket is closed.

Socket Example

Consider a very simple socket example.

The client sends a number to the server. The server calculates the square of that number and returns the result to the client.

The following pseudocode represents the application-layer code used by the server and client. The programs use transport-layer operations to transmit the messages.

The transport-layer operations are:

Client Side

- **CONNECT(address, destinationPort):** Requests a connection to the specified host. This operation is used only with connection-oriented protocols such as TCP. It returns the socket created for the connection.
- **SEND(socket, data):** Sends data through the connection or port using TCP or UDP.
- **RECEIVE(socket):** Receives data through the socket using TCP or UDP.

- **CLOSE(socket):** Terminates a TCP connection.

Server Side

- **WAIT(localPort):** Waits for incoming connection requests on the specified port. This operation is used only with TCP and returns the socket created for the accepted connection. It is a simplified representation of the operations through which a server begins listening and accepts a connection.
- **SEND(socket, data):** Sends data through the connection or port using TCP or UDP.
- **RECEIVE(socket):** Receives data through the socket using TCP or UDP.
- **CLOSE(socket):** Closes the connection or communication endpoint.

The following examples show the client and server sockets.

Socket client

```
serverAddress = "127.0.0.1"
serverPort = 65432
number = 5

socket = CONNECT(serverAddress, serverPort)
SEND(socket, number)
result = RECEIVE(socket)
CLOSE(socket)
```

Socket server

```
serverPort = 65432

REPEAT:  # After completing an operation, the server listens again
    socket = WAIT(serverPort)
    data = RECEIVE(socket)
    result = data * data
    SEND(socket, result)
    CLOSE(socket)
```